Towards Designing SVM Coherence Protocols Using High-level Specifications and Aspect-oriented Translations

David Mentré, Daniel Le Métayer, Thierry Priol

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

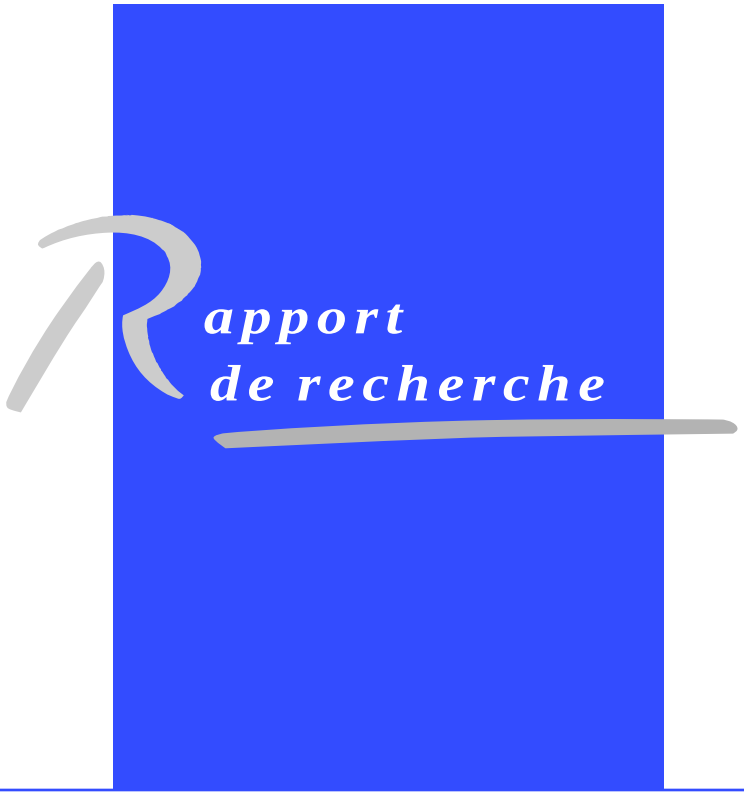
Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations

David Mentré, Daniel Le Métayer, Thierry Priol

N°3765

Septembre 1999

THÈME 1



ISSN 0249-6399

Towards designing SVM coherence protocols using high-level specifications and aspect-oriented translations

David Mentré, Daniel Le Métayer, Thierry Priol

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n ° 3765 — Septembre 1999 — 13 pages

Abstract: We propose a two-stage approach for the design of a shared virtual memory. The first stage is a high-level protocol description using the Structured Gamma formalism which is amenable to formal verifications. The second is a translation of this abstraction into an automaton which can be dynamically loaded in a runtime environment that provides a concept of global directory. This translation is based on Aspect-Oriented Programming which makes it possible to specify independently different aspects of the implementation. The complete system can thus be easily ported on a new environment.

Key-words: Shared Virtual Memory (SVM), coherence protocol, specification, Gamma, aspects.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Conception de protocoles de cohérence de MVP par traduction d'une spécification Gamma à l'aide d'aspects

Résumé : Nous proposons un cadre permettant de concevoir des mémoires virtuelles partagées à l'aide d'une spécification de protocoles de cohérence. La conception s'effectue en plusieurs phases. La première est une description de haut niveau dans une variante du formalisme Gamma Structuré qui permet d'effectuer certaines vérifications. La seconde consiste à traduire cette abstraction en automate chargeable au sein d'un environnement d'exécution supportant le concept de répertoire global. Cette traduction est réalisée par une technique de programmation par « aspects » qui permet de spécifier indépendamment certaines caractéristiques de la mise en œuvre (contrôle et représentation des données). L'ensemble du système peut ainsi être facilement porté dans un nouvel environnement.

Mots-clé : Mémoire Virtuelle Partagée (MVP), protocole de cohérence, spécification, Gamma, aspects.

1 Introduction

Distributed systems made of autonomous machines having memory and processors linked by a network are widely used. However, there programming is not a simple task as one must explicitly deal with resource distribution. To ease this programming, the Shared Virtual Memory (SVM) concept has been proposed by Li and Hudak [10]. This concept offers the illusion of a global address space over a distributed address space. To improve performance of such a system, several copies of a piece of data are made using caches. Therefore a cache coherence protocol is needed to guarantee a coherent view of the system.

Since the first protocol, several ones have been proposed like the lazy release consistency [7] or the scope consistency [6]. However a protocol aims at optimizing a limited range a application sharing patterns. Therefore a general purpose SVM must integrate several protocols. However this static approach cannot take into account new technology like Remote Memory Access [5]. Therefore protocols should be dynamically added in the SVM.

Moreover a SVM must be tighlty integrated with the operating system to achieve optimal performance [9]. But as a system component, a faulty implementation of a SVM can have disastrous effect. An approach to increase confidence level in a SVM component is the use of a high-level formalism for the SVM design. This abstraction helps the designer focus on the coherence protocol logic without dealing with low-level implementation details. Moreover, two advantages are gained with a formal abstraction of the protocol. Firstly, with an adequate system abstraction, some automatic verification can be done [11]. Secondly, with a proper translation mechanism to low-level implementation, the same protocol can be targeted to several interconnection technologies. And if the protocol environment is sufficiently small and well defined, the whole SVM component can be easily ported on a new system.

This paper presents the state of our current research on the design of programmable SVM. We propose to define coherence protocols in two parts. The first is a high level-level description free of any implementation detail. The second one defines a transformation into an automaton following certain implementation choices. This automaton can be loaded dynamically into a runtime.

We present in section 2 other programmable coherence protocol design environments and their limitations. In section 3 we give a global overview of our approach. Then in section 4, we present the Li and Hudak coherence protocol which is used throughout this paper to illustrate our approach. We then show in section 5 how a coherence protocol is formalized. In section 6, we describe the translation process to produce an implementation from an abstract protocol description. This implementation is loaded into a runtime environment described in section 7. We finally conclude in section 8 by giving current state of our work and perspectives.

2 Programmable Shared Virtual Memories

The idea of programmable SVM is not new and we are aware of at least two efforts of such environments, PCS and TeaPot. Each of them proposes a specific method differing from the other one by the programming language used, the programming style and facilities offered by the protocol programming environment.

PCS [12] is a runtime environment for distributed filesystems. This a not strictly speaking a SVM programming environment but issues are identical. Protocols are described in a Tcl like language and executed in an interpreter loaded as a Mach 3 external memory manager. Several facilities are offered to ease protocol programming: messages are kept in order and temporary states due to Mach 3 kernel and message waiting between two states are eliminated. As protocols are interpreted, the environment is very slow. Moreover no protocol verification is proposed.

TeaPot [3] is a SVM programming environment. Protocols are described as automata in a dedicated language. Each state describes messages that can be received and actions to be done. The environment offers some facilities: undesired message wait queue, a continuation mechanism to factorize intermediate states and messages are kept in order. A protocol can be verified with state enumeration (MurΦ verifier). Due to combinatory explosion, only protocol involving two nodes can be verified.

Those environments have limits. Firstly, formalism used to describe protocols is the low-level implementation automata. Therefore the logic behind a protocol is hard to understand and protocol writing and debugging is painful. Secondly, verification is either not available (PCS) or limited (TeaPot). Thirdly, those environments are bound to a network technology (message-passing for TeaPot) and thus protocols cannot be adapted to a new technology. To answer those limits we propose a new approach.

3 Global Overview

We consider a parallel system with distributed memory. Our objective is to design a programmable SVM using the underlying paging mechanism. Detection and data management granularity is thus a page. Our goal is to propose a method to design SVM from high-level protocol specifications. This approach enforces a clean separation between coherence choices related to a protocol and implementation choices related to a network technology. Protocol design and checking is thus made easier. Our work is based on an elaboration of the Structured Gamma formalism [4] that we use as a specification language. Its main advantage is that it allows automatic invariant checking on high-level descriptions. Moreover this checking does not depend on the number of nodes involved in the distributed system, thus avoiding the combinatory explosion of state enumeration. Furthermore communications are not specified at this level. As a consequence, many implementations can be built from the same abstract protocol. Finally, the explicit definition of invariants improves the understanding of the protocol.

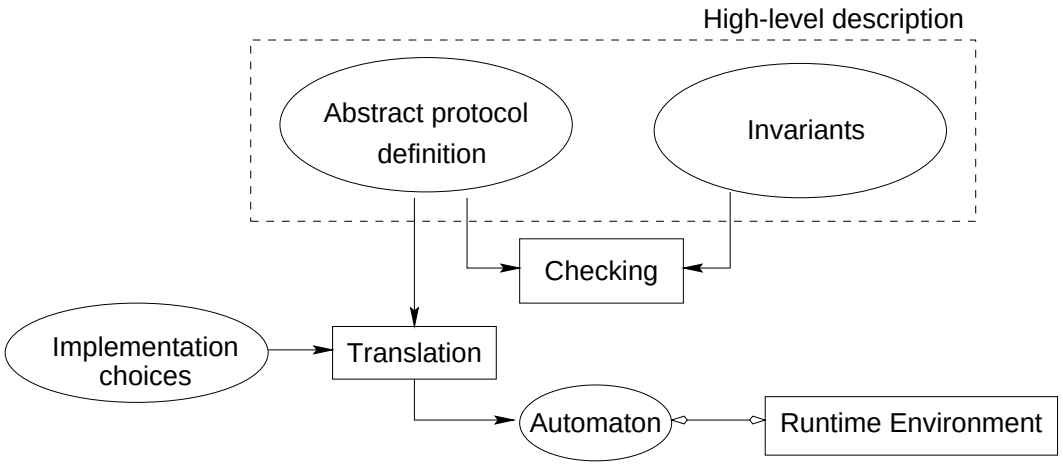


Figure 1: Programming environment

Figure 1 summarizes our approach. First of all, the programmer introduces a *high-level description* of the protocol. This description is made up of an *abstract protocol definition* describing state changes and *invariants*. These two parts allow protocol *checking*. Then a *translation* stage transforms the *abstract protocol definition* into an *automaton*. This *translation* is based on *implementation choices* (available as Aspects [8]) made by the programmer. Finally the resulting *automaton* can be loaded within a *runtime environment* and interacts with it. We now describe the protocol used as an example throughout this article.

4 The Li and Hudak Coherence Protocol

We use the Li and Hudak single-writer/multiple-readers coherence protocol [10] as an example to illustrate our approach. This protocol is simple and concise and thus can be described in this paper. However our approach is suitable to formalize any type of coherence protocol like eager release consistency [2]. The Li and Hudak protocol handle each memory page of the shared address space independently. A page is either written by a unique node or read by one or more nodes. Each reading node makes a local copy of the page. When a page in read state is written, all copies of the page must be deleted (invalidation phase). Figure 2 shows the state graph of a page. State change is triggered either by a local or a remote access. For each page, a manager stores the page state and copies knowledge. Access requests are sent to it and it triggers needed invalidations.

We now show how a protocol can be described in our high-level formalism.

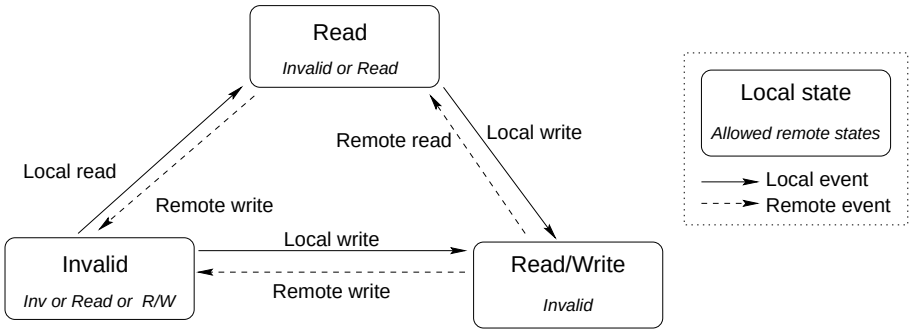


Figure 2: Page state graph of Li and Hudak coherence protocol

5 Protocol description

Our protocol specification language is an elaboration of the Structured Gamma formalism [4]. We detail this formalism and then apply it to the Li and Hudak protocol.

5.1 The Structured Gamma Formalism

The Structured Gamma formalism [4] is based on the chemical reaction metaphor. The unique data structure in Gamma [1] is the multiset (a set than can contain several occurrences of the same element) which can be seen as a "chemical solution". A simple program is a pair (*Reaction condition*, *Action*). Execution proceeds, without an explicit order, by replacing elements in the multiset satisfying the reaction condition by the products of the action ("chemical reaction"). The result is obtained when a stable state is reached, that is to say when no more reactions can take place. Locality and lack of ordering allow a description of algorithms without unnecessary sequentiality. An example of a Gamma program is factorial: $[x, y \rightarrow x \times y]$. Applied to multiset $\{1, 2, 3, 4\}$ it could produce the following execution sequence:

$$[1, 3 \rightarrow 3] \text{ simultaneously with } [2, 4 \rightarrow 8] \text{ then } [3, 8 \rightarrow 24]$$

The Structured Gamma formalism [4] introduces the notion of typed multiset. Types are based on relations between the elements of a multiset. More precisely, a type imposes restrictions on the number of occurrences of values satisfying specific relations. A checking algorithm allows to demonstrate that a set of Gamma rules satisfy a specific type. It is wise to notice that the addition of such a type does not invalidate the locality property of Gamma rules. In our context, Gamma rules are used to specify a coherence protocol and a Structured Gamma type defines invariants of this protocol. A type checking algorithm allows us to verify that a protocol satisfy a type (that is to say an invariant).

Protocol Gamma Rules :

R_1 : **ReadDetect** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **Read** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_1$, **PageFrame** $pf_2\ p\ n_2$

R_2 : **ReadDetect** $p\ n_1$, **ReadWrite** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **Read** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_1$, **PageFrame** $pf_2\ p\ n_2$

R_3 : **WriteDetect** $p\ n_1$, **ReadWrite** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
→ **ReadWrite** $p\ n_1$, **PageFrame** $pf_2\ p\ n_1$

R_4 : **WriteDetect** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$, **Ok** p
→ **InvalidationPhase** $pf_2\ p\ n_1$

R_5 : **InvalidationPhase** $pf_1\ p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_1\ p\ n_2$
→ **InvalidationPhase** $pf_1\ p\ n_1$

R_6 : **InvalidationPhase** $pf_1\ p\ n_1$, ¬**Read** $p\ n_2$
→ **ReadWrite** $p\ n_1$, **PageFrame** $pf_1\ p\ n_1$, **Ok** p

R_7 : **WriteDetect** $p\ n_1$, **Read** $p\ n_1$, **PageFrame** $pf_1\ n_1$, **Ok** p
→ **InvalidationPhase** $pf_1\ p\ n_1$

Invariants :

I_1 : $\overline{\textbf{ReadWrite}}\ p\ * \leq 1$
I_2 : $\overline{\textbf{Read}}\ p\ * \geq 1 \Rightarrow \overline{\textbf{ReadWrite}}\ p\ * = 0$
I_3 : $\overline{\textbf{ReadWrite}}\ p\ * = 1 \Rightarrow \overline{\textbf{Read}}\ p\ * = 0$
I_4 : $\overline{\textbf{PageFrame}}\ *\ p\ n = \overline{\textbf{Read}}\ p\ n + \overline{\textbf{ReadWrite}}\ p\ n$
I_5 : $\overline{\textbf{ReadDetect}}\ p\ n + \overline{\textbf{WriteDetect}}\ p\ n \leq 1$
I_6 : $\overline{\textbf{InvalidationPhase}}\ *\ p\ n > 0 \Rightarrow \overline{\textbf{ReadWrite}}\ p\ n = 0$
I_7 : $\overline{\textbf{PageFrame}}\ pf\ p\ * \geq 0 \Rightarrow \overline{\textbf{PageFrame}}\ pf\ p\ * = \overline{\textbf{PageFrame}}\ *\ p\ *$
I_8 : $\overline{\textbf{Read}}\ p\ * + \overline{\textbf{ReadWrite}}\ p\ * \geq 1$

Figure 3: Structured Gamma version of single-writer/multiple-readers Li and Hudak protocol

5.2 Formalization of the Li and Hudak Protocol

Figure 3 gives the complete Structured Gamma specification of the Li and Hudak protocol. Rules defining the protocol are numbered R_1 to R_7. Constraints defining the invariant are labeled I_1 to I_8. A Gamma rule defines a protocol state change. Those rules use relations which map to physical objects (page table, page content), exceptions or messages. More precisely, **ReadWrite** $p\ n$ relation states that page p on node n is in read/write mode (page table rights); **Read** $p\ n$ relation states a read right; **ReadDetect** $p\ n$ (respectively

WriteDetect p n) relation states that a read (respectively write) exception has occured for page p on node n; **PageFrame** pf p n relation states that a page frame pf (a physical page) corresponding to virtual page p exists on node n; **InvalidationPhase** pf p n relation states that page p is in invalidation phase in order that page frame pf will be accessible in read/write mode on node n; **Ok** p relation states that page p is not in invalidation phase. Implementation (in arrays, booleans, signals, ...) of those relations is defined by the aspects.

To delete or to add a relation in a rule details how is made the corresponding state change. For example, **ReadWrite** p n relation in left hand side but not in right hand side of a rule states that the read/write right has been lost for page p on node n. Rules R_1 and R_2 state case where node n_1 requests read access for a page and this page is currently used in read (R_1) or write (R_2) mode by another node n_2. Rules R_3, R_4 and R_7 state case where node n_1 makes a write on a page and this page is (a) either accessed with write (R_3) or read (R_4) by another node n_2 or (b) accessed by itself (node n_1) in read mode (R_7). Finally, rules R_5 and R_6 state respectively the invalidation loop of page copies and the grant of read/write access when all invalidations are made. One should notice that rule R_6 contains a negation. This change to the pure Gamma formalism (which is without global condition) ease protocol description. Moreover this extension does not impact verification and translation.

In invariant definition, an overligned relation like $\overline{\textbf{ReadWrite}}$ p $*$ expresses the occurrence number of n-uplets satisfying relation with given arguments ($*$ can be bound to any value). Constraint $\overline{\textbf{ReadWrite}}$ p $* \leq 1$ states that for a page p, there is at most one node n such that **ReadWrite** p n exists. Within the protocol, it is equivalent to say that at any time, at most one node (node n) can write on page p. Constraint I_1 states that at most one node can have read/write access on a page. I_2 states that if a node has a read access on a page, no other one have write access to it. I_3 states the reverse case. I_4 (combined with I_2 and I_3) states that for each page in read or read/write mode, a node must have a copy of it (a page frame). I_5 checks that a read and a write page faults cannot be simultaneously detected on a page. I_6 states that in invalidation phase, the node having triggered invalidation should not have read/write access to this page. I_7 states that if there is more than one copy of a page, they are the same. Finally I_8 checks that at least a copy of each page exists in the system.

Invariant checking is done by considering the effect of each rule over the occurence number of each relation. For example, consider I_1. The sole rule that impacts $\overline{\textbf{ReadWrite}}$ p $*$ are rules R_2 and R_6 (because they have a different occurence number of **ReadWrite** in left and right hande side). Rule R_2 decreases $\overline{\textbf{ReadWrite}}$ p $*$ by a unit while R_6 increments it. Therefore I_1 is trivially satisfied by all rules except R_6. But R_6 satisfies a I_6 premise ($\overline{\textbf{InvalidationPhase}}$ $*$ p $n > 0$). Thus, before R_6 application, $\overline{\textbf{ReadWrite}}$ p $n = 0$ and $\overline{\textbf{ReadWrite}}$ p $n = 1$ after; I_1 is thus verified in any case.

6 Protocol translation

We now show an overview of the translation process and its application to rule R_1 of the protocol.

6.1 Translation with aspects

Translation of Gamma rules into an automaton is performed using a technique inspired by Aspect-Oriented Programming [8]. It makes it possible to separately define "aspects" that are usually spread throughout the program (like synchronization, communication or memory management). Usual programming does not offer any mean to group them in autonomous entities (function, module or object). Using Aspect-Oriented programming, a program is made of a base program and one or more aspects that set cross-program properties. A weaver merges properties described in those aspects within the base program. To transform a high-level protocol into an automaton we use two aspects describing respectively *control* and *data representation*. They produce elementary actions which use functionalities of the runtime environment. A compilation step transforms them into an *automaton* which can be loaded into the system.

6.2 Translation of rule R_1

We now show how rule R_1 of the Gamma formalization of the Li and Hudak protocol is translated into its low-level counter part. The first aspect, *control*, gives a hint about how the rule can be inserted in the runtime environment. For R_1 it is written "`triggers: ReadDetect`". It states that **ReadDetect** is the triggering relation in that rules, that is to say that introduction of this element in the multiset will start other rule relations' evaluation.

The second aspect, *data representation*, states how relations are implemented within the runtime environment. An excerpt of this aspect needed to translate R_1 is given in figure 4. Each translation rule follows the pattern "*Pattern* `==>` *Actions to introduce*". For each pattern, the aspect states actions to do. Actions use primitives from the runtime environment. A "`?(Relation a b)`" pattern states actions to execute in order to check the existence of this relation. A "`Relation a b -> not(Relation a b)`" pattern states actions to execute if a relation exists in left hande side but not right hand side of a rule. The reverse for "`not(Relation a b) -> Relation a b`". For each pattern, there is a set of actions which are of two kinds; *test* (written "`(` *code* `)`") and state change (written "`{` *code* `}`"). A test makes actions following it under its condition. Actions code is written in a classic language (in our example C++).

The translation process tests successively the pattern of each transformation as they appear in the aspect. If a match occurs, transformation actions are added to the resulting code with variables substituted. Thus the order of transformations within the aspect determines the sequential order of actions in the resulting code. The list of actions from R_1 translation is given in figure 5.

7 Runtime Environment

The runtime environment is composed of two parts: an *automaton engine* (cf. figure 6, [a]) and a *global directory* [b]. The automaton engine is actived when an event is received, it determines actions to execute with internal states and then executes those actions.

```
?(ReadDetect p n)
==>
( event_match(read_page_fault, &p, &n) ) ;

?(Read p n)
==>
( copy_set->seek(&p, &n, read) ) ;

not(PageFrame pf p n) ,
not(InvalidationPhase pf p n)
  --> PageFrame pf p n
==>
{ memory->page_alloc(n, p); } ;

not(Read p n) --> Read p n
==>
{ memory->access(n, p, read); }
{ copy_set->set(p, n, read); } ;

PageFrame pf2 p n2 --> PageFrame pf2 p n1
==>
{ memory->page_copy(n2, p, n1, p); } ;
```

Figure 4: Data representation aspect excerpt needed for R_1

R_1 : **ReadDetect** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_2$
$\rightarrow$ **Read** $p\ n_1$, **Read** $p\ n_2$, **PageFrame** $pf_2\ p\ n_1$, **PageFrame** $pf_2\ p\ n_2$

Result code	Meaning
`( event_match(read_page_fault, &p, &n1) )`	Read page fault received for page p on node n_1
`( copy_set->seek(&p, &n2, read) )`	In table `copy_set`, look for another node n_2 where page p is in read mode (`read` state)
`( n1 != n2 )`	Check that n_1 and n_2 are different
`{ memory->page_alloc(n1, p); }`	Page-frame allocation on node n_1 for page p
`{ memory->access(n1, p, read); }`	Page p on node n_1 gains `read` access right
`{ copy_set->set(p, n1, read); }`	Store in `copy_set` table access right modification
`{ memory->page_copy(n2, p, n1, p);}`	Physical copy from page p on n_2 to page p on n_1

Figure 5: Rule R_1 and translated code

Actions use a global directory to modify system states [c] (address spaces, page access rights, etc.) and to store protocol information [d]. This global directory is seen identically by each node. It triggers communication and data exchanges between nodes. This global directory can be implemented in various ways depending on the available network technology.

8 Conclusion and Perspectives

We have introduced a framework for the description of SVM protocols and their implementation. It is based upon the Structured Gamma formalism for abstract protocol description and upon the Aspect-oriented inspired technique to translate this abstraction into a concrete implementation. Two aspects, control and data representation, are used to achieve this translation.

The implementation of this framework is currently in progress using a cluster of SMP x86 workstations running the Linux operating system and interconnected using the Scalable Coherent Interface (SCI). On the verification side, the checking algorithm sketched in section 5.2 should be described formaly and proved.

In the future, this programmable SVM system will be used to compare protocols on the same architecture. This would allow us to extract protocol profiles in order to make a

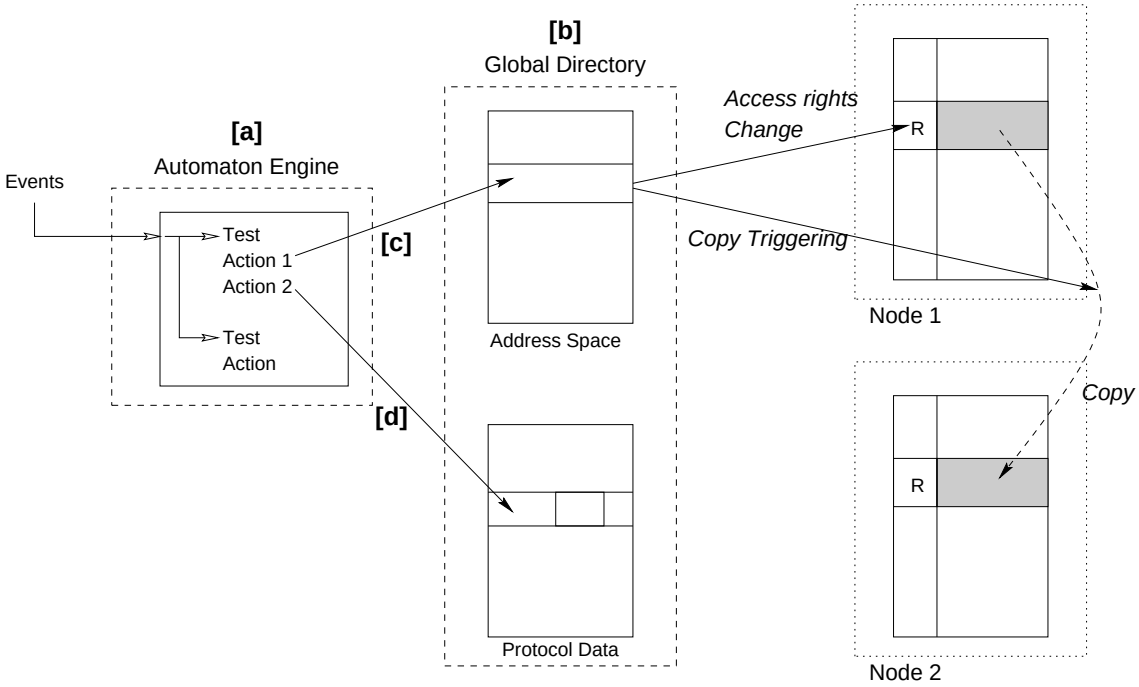


Figure 6: Protocol execution environment

more systematic (or even automatic) choice of protocol for a given data sharing pattern of a parallel application.

References

[1] J.-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, Jan. 1993.

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 152–64. Association for Computing Machinery SIGOPS, Oct. 1991.

[3] S. Chandra, B. Richards, and J. R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[4] P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, July 1998.

[5] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–21, Feb. 1992.

[6] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Anual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.

[7] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, Jan. 1994.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer.

[9] Z. Lahjomri and T. Priol. KOAN: A shared virtual memory for the iPSC/ 2 hypercube. *Lecture Notes in Computer Science*, 634:441–??, 1992.

[10] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors* . Ph.D. Thesis, Yale University, Sept. 1986.

[11] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formals Methods: A Practical Tool for OS Implementors. In *Proceedings of the 6th IEEE Workshop on Hot Topics in Operating Systems*, May 1997.

[12] K. Uehara, S. Inohara, H. Miyazawa, and K. Yamamoto. A Framework for Customizing Coherence Protocols of Distributed File Caches. In *Proceedings of IEEE 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 83–90, May 1996.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399